Runtime Enforcement of K-step Opacity

Yliès Falcone, Hervé Marchand

Runtime Enforcement of K-step Opacity*

Yliès Falcone Hervé Marchand

***Abstract*— We study the enforcement of K-step opacity at runtime. In K-step opacity, the knowledge of the secret is of interest to the attacker within K steps after the secret occurs and becomes obsolete afterwards. We introduce the mechanism of runtime enforcer that is placed between the output of the system and the attacker and enforces opacity using delays. If an output event from the system violates K-step opacity, the enforcer stores the event in the memory, for the minimal number of system steps until the secret is no longer interesting to the attacker (or, K-step opacity holds again).**

I. Introduction

Security is a major concern in nowadays information systems. Among existing security notions, *opacity* (see e.g., [2], [3]) is a generic and general notion used to express several existing confidentiality concerns such as trace-based non-interference and anonymity (cf. [3]) and even secrecy (cf. [4]). Opacity aims at preserving unwanted retrievals of a system secret (e.g., values of confidential variables) by untrustworthy users while observing the system. When examining the opacity of a secret on a given system, we check whether there are some executions of the system which can lead an external attacker to know the secret; in that case the secret is said to be *leaking*. While usual opacity is concerned by the *current* disclosure of a secret, K-step opacity, introduced in [5], additionally models secret retrieval in the past (e.g., K execution steps before).

Ensuring opacity on a system is usually performed using *supervisory control* (cf. [6], [7], [8], [9]). Supervisory control consists in using a so-called *controller* to disable undesired behaviors of the system, e.g., those leading to reveal the secret. Moreover, the technique of *dynamic observability* was proposed in [10] to ensure opacity by dynamically restraining, at each system step, the set of observable events. Finally, [11] enforces opacity by statically inserting additional events in the output behavior of the system.

These techniques suffer from practical limitations preventing their applicability in some situations. Static techniques such as supervisory control are *intrusive* which entail to *disable some (internal) behaviors* of the underlying system. While ensuring opacity via dynamic observability comes at the price of *destroying observable behavior* of the system. Those limitations motivate for investigating the use of other validation techniques to ensure opacity.

We are interested in runtime validation techniques, namely *runtime verification* and *runtime enforcement*, so as to validate several levels of opacity on a system. Runtime verification (cf. [12], [13], [14]) consists in checking during the execution of a system whether a desired property holds or not. Generally, one uses a special decision procedure, a *monitor*, grabbing information in the run of an executing system and acting as an oracle to decide property validation or violation. Runtime enforcement (see [15], [16], [17]) is an extension of runtime verification aiming to circumvent property violations. Within this technique the monitor not only observes program executions, but also modifies them, using an internal memorization mechanism.

II. The proposed approach

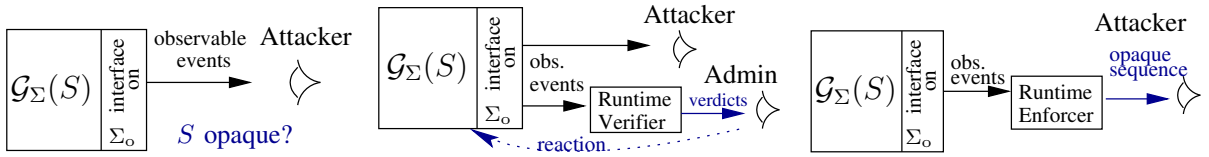
The problem can be depicted in Fig. 1a. A system $\mathcal{G}$ produces sequences of events belonging to an alphabet Σ. Among the possible executions of the system, some of these are said to be *secret*. Some events of the system, in a sub-alphabet $\Sigma_o \subseteq \Sigma$, are observable by an external attacker through the system interface. We assume that the attacker does not interfere with the system (i.e., he performs only observations through the interface) and has a perfect knowledge of the structure (even internal) of the system. We are interested in the opacity of the secret executions on the considered system. That is, from a sequence of observable events, the attacker should not be able to deduce whether (a prefix of) the current execution of the system (corresponding to this observation) is secret or not. In this case, the secret S is said to be *opaque* wrt. the considered system and its interface.

We now sketch the techniques that we propose to analyze and validate opacity. When model-checking (Fig. 1a) the opacity of a secret on the system, we take its specification to perform an analysis that provides the executions leading to a security leakage when they are observed through the interface of the system. This indicates existing flaws to the system designer. When verifying opacity at runtime (Fig. 1b), we introduce a runtime verifier which observes the same sequence of observable events as the attacker and produces verdicts related to the preservation or violation of the opacity. With such a mechanism, the system administrator may react and (manually) take appropriate measures. When enforcing opacity at runtime (Fig. 1c), we introduce a runtime enforcer between the system and the attacker. The sequence of observable events is directly fed to the enforcer. The attacker now observes the outputs produced by the runtime enforcer. The runtime enforcer modifies its input sequence and produces a new one in such a way that, on the output execution sequence seen by the attacker, opacity is preserved wrt. the actual execution on the initial system. Thus, the proposed runtime enforcement automatically prevents opacity violation.

These runtime validation techniques have several advantages. First, these techniques are *not intrusive*. Indeed, a runtime verification or runtime enforcement framework does

*An extended version of this paper with proofs is available as [1].

Y. Falcone is with University Grenoble 1 and LIG, Grenoble, France, H. Marchand is with Inria Rennes Bretagne-Atlantique, Rennes, France.



(a) System vs Opacity (b) Opacity Verification (c) Opacity Enforcement

Fig. 1. Several ways to validate the opacity of a secret on a system

not suppose being able to modify the internal behavior of the monitored system. It is particularly useful when dealing with legacy code, leading model-checking to become obsolete since the internal system behavior cannot be modified. Moreover, the proposed runtime-based approaches do not *distort* the internal nor the observable behavior of the system. Furthermore, checking the opacity at runtime allows to react to misbehaviors; as shown with runtime enforcement in this paper. Ensuring opacity with runtime enforcement also has the advantage to modify the observable behavior of the system *in a minimal way*: our runtime enforcers minimally delay the initial execution sequence.

III. Preliminaries and Notation

A. Execution sequences

Unless otherwise specified, considered functions are total. $\mathbb{N}$ denotes the set of non-negative integers. Considering a finite set of elements E, a *sequence* s over E is a function $s : I \to E$ where I is the integer interval $[0, n]$ for some $n \in \mathbb{N}$. A *language* over E is a set of sequences over E. We denote by E^* the universal language over E (i.e., the set of all finite sequences over E), by E^+ the set of non-empty finite sequences over E. Furthermore, for $n \in \mathbb{N}\setminus\{0,1\}$, the generalized Cartesian product of E is $E^n \stackrel{\text{def}}{=} E\times E\times\cdots\times E$, i.e., the Cartesian product of E of dimension n. The empty sequence of E^* is denoted by ϵ_E or ϵ when clear from the context. The length of a finite sequence $s \in E^*$ is noted $|s|$. For a sequence $s \in E^+$ and $i < |s|$, the $(i+1)$-th element of s is denoted by s^i, and the subsequence containing the $i+1$ first elements of s is denoted $s^{\cdots i}$. For $s, s' \in E^*$, we denote by $s\cdot s'$ the concatenation of s and s', and by $s \preceq s'$ the fact that s is a prefix of s' (i.e., when $\forall i \in [0, |s|-1] : s^i = s'^i$ and $|s| \leq |s'|$). The *prefix-closure* of a language L wrt. E^* is defined as $\text{Pref}(L) \stackrel{\text{def}}{=} \{s \in E^* \mid \exists s' \in E^* : s\cdot s' \in L\}$. Given $s' \preceq s$, $|s - s'| \stackrel{\text{def}}{=} |s| - |s'|$.

Behaviors of systems are modeled by Labelled Transitions Systems (LTS for short) whose actions belong to a finite set Σ. Sequences of actions are named *execution sequences*. The formal definition of an LTS is as follows:

Definition 1 (LTS): A deterministic LTS is a 4-tuple $\mathcal{G} = (Q^\mathcal{G}, q^\mathcal{G}_{\text{init}}, \Sigma, \delta_\mathcal{G})$ *where* $Q^\mathcal{G}$ *is a finite set of states,* $q^\mathcal{G}_{\text{init}} \in Q^\mathcal{G}$ *is the initial state,* Σ *is the alphabet of* actions*, and* $\delta_\mathcal{G} : Q^\mathcal{G} \times \Sigma \to Q^\mathcal{G}$ *is* the partial transition function.

We consider an LTS $\mathcal{G} = (Q^\mathcal{G}, q^\mathcal{G}_{\text{init}}, \Sigma, \delta_\mathcal{G})$. We write $q \xrightarrow{a}_\mathcal{G} q'$ for $\delta_\mathcal{G}(q, a) = q'$ and $q \xrightarrow{a}_\mathcal{G}$ for $\exists q' \in Q^\mathcal{G} : q \xrightarrow{a}_\mathcal{G} q'$. We extend $\to_\mathcal{G}$ to arbitrary execution sequences by setting: $q \xrightarrow{\epsilon}_\mathcal{G} q$ for every state q, and $q \xrightarrow{s\sigma}_\mathcal{G} q'$ whenever $q \xrightarrow{s}_\mathcal{G} q''$ and $q'' \xrightarrow{\sigma}_\mathcal{G} q'$, for some $q'' \in Q^\mathcal{G}$. For any language $L \subseteq \Sigma^*$ and set of states $X \subseteq Q^\mathcal{G}$, we set $\Delta_\mathcal{G}(X, L) \stackrel{\text{def}}{=} \{q \in Q^\mathcal{G} \mid \exists s \in L, \exists q' \in X : q' \xrightarrow{s}_\mathcal{G} q\}$. $\mathcal{L}(\mathcal{G}) \stackrel{\text{def}}{=} \{s \in \Sigma^* \mid q^\mathcal{G}_{\text{init}} \xrightarrow{s}_\mathcal{G}\}$ denotes the set of execution sequences of $\mathcal{G}$. Given a set of marked states $F_\mathcal{G} \subseteq Q^\mathcal{G}$, the *marked language* of $\mathcal{G}$ is $\mathcal{L}_{F_\mathcal{G}}(\mathcal{G}) \stackrel{\text{def}}{=} \{s \in \Sigma^* \mid \exists q \in F_\mathcal{G} : q^\mathcal{G}_{\text{init}} \xrightarrow{s}_\mathcal{G} q\}$, i.e., the execution sequences that end in $F_\mathcal{G}$. Notations apply to finite-state machines which are LTSs with an output function.

B. Observational behavior

The observation interface between a user and the system is specified by a sub-alphabet of events $\Sigma_o \subseteq \Sigma$. The user observation through an interface is then defined by a *projection*, denoted by P_{Σ_o}, from Σ^* to Σ_o^* that erases in an execution sequence of Σ^* all events not in Σ_o. Formally, $P_{\Sigma_o}(\epsilon_\Sigma) \stackrel{\text{def}}{=} \epsilon_\Sigma$ and $P_{\Sigma_o}(s\cdot\sigma) \stackrel{\text{def}}{=} P_{\Sigma_o}(s)\cdot\sigma$ if $\sigma \in \Sigma_o$ and $P_{\Sigma_o}(s)$ otherwise. This definition extends to any language $L \subseteq \Sigma^*$: $P_{\Sigma_o}(L) \stackrel{\text{def}}{=} \{\mu \in \Sigma_o^* \mid \exists s \in L : \mu = P_{\Sigma_o}(s)\}$. In particular, given an LTS $\mathcal{G}$ over Σ and a set of observable actions $\Sigma_o \subseteq \Sigma$, the set of *observed traces* of $\mathcal{G}$ is $\mathcal{T}_{\Sigma_o}(\mathcal{G}) \stackrel{\text{def}}{=} P_{\Sigma_o}(\mathcal{L}(\mathcal{G}))$. Given two execution sequences $s, s' \in \Sigma^*$, they are equivalent w.r.t. P_{Σ_o}, noted $s \approx_{\Sigma_o} s'$ whenever $P_{\Sigma_o}(s) = P_{\Sigma_o}(s')$. Given two execution sequences s, s' such that $s' \preceq s$, $s \setminus s'$ is the suffix of s that permits to extend s' to s, and $|s - s'|_{\Sigma_o} \stackrel{\text{def}}{=} |P_{\Sigma_o}(s)| - |P_{\Sigma_o}(s')|$ corresponds to the number of observable events that are necessary to extend s' into s. Conversely, given $L \subseteq \Sigma_o^*$, the *inverse projection* of L is $P^{-1}_{\Sigma_o}(L) \stackrel{\text{def}}{=} \{s \in \Sigma^* \mid P_{\Sigma_o}(s) \in L\}$. Given $\mu \in \mathcal{T}_{\Sigma_o}(\mathcal{G})$, $[\![\mu]\!]^\mathcal{G}_{\Sigma_o} \stackrel{\text{def}}{=} P^{-1}_{\Sigma_o}(\mu) \cap \mathcal{L}(\mathcal{G})$ (noted $[\![\mu]\!]_{\Sigma_o}$ when clear from context) is the set of observation traces of $\mathcal{G}$ *compatible* with μ, i.e., execution sequences of $\mathcal{G}$ having trace μ. Given $\mu' \preceq \mu$, we note $[\![\mu'/\mu]\!]_{\Sigma_o} \stackrel{\text{def}}{=} [\![\mu']\!]_{\Sigma_o} \cap \text{Pref}([\![\mu]\!]_{\Sigma_o})$ the set of traces of $\mathcal{G}$ that are still compatible with μ' knowing that μ' is the prefix of μ that occurred in the system.

C. K-delay state estimators

To generate runtime verifiers and enforcers, we will need the notion of K-delay state estimator introduced in [5]. Intuitively, a K-delay state estimator, according to the observation interface of a system, indicates the estimated states of the system during the K previous steps.

We consider the set of l-tuples of states, for $l \geq 2$, of $\mathcal{G}$. Elements of Q^l model partial sequences of states. A set $m \in 2^{Q^l}$ is called an l-dimensional state mapping. We denote by $m(i)$ the set of the $(l-i)^{\text{th}}$ state of elements of m. Intuitively, $m(0)$ corresponds to the current state estimate whereas $m(i)$ corresponds to the state estimate knowing that i observations have been made. We also need to define:

- the *shift operator* $\blacktriangleleft$: $2^{Q^l} \times 2^{Q^2} \to 2^{Q^l}$ s.t.

$$m \blacktriangleleft m_2 \stackrel{\text{def}}{=} \{(q_2, \ldots, q_{l+1}) \in Q^l \mid (q_1, \ldots, q_l) \in m \wedge (q_l, q_{l+1}) \in m_2\},$$

- the observation mapping $\mathrm{Obs} : \Sigma_\mathrm{o} \to 2^{Q^2}$ s.t.

$$\mathrm{Obs}(\sigma) \stackrel{\text{def}}{=} \{(q_1, q_2) \mid \exists s \in \Sigma^+ : P_{\Sigma_\mathrm{o}}(s) = \sigma \wedge q_1 \stackrel{s}{\to}_\mathcal{G} q_2\},$$

- the function $\odot_l : 2^E \to 2^{E^l}$ s.t. $\odot_l(E) \stackrel{\text{def}}{=} \{(e, \ldots, e) \mid e \in E\}$, for a set E.

Based on the previous operations, K-delay state estimators are defined as follows:

Definition 2 (K-Delay State Estimator): For $\mathcal{G}$, a secret S, a projection $\mathcal{P}_{\Sigma_\mathrm{o}}$, the K-delay state estimator *is an LTS $D = (M^D, m^D_{\text{init}}, \Sigma_\mathrm{o}, \delta_D)$ s.t.:*

- *M^D is the smallest subset of $2^{Q^{K+1}}$ reachable from m^D_{init} with δ_D,*
- $m^D_{\text{init}} \stackrel{\text{def}}{=} \{t_q \in \odot_{K+1}(\{q\}) \mid q \in \Delta_\mathcal{G}(\{q^\mathcal{G}_{\text{init}}\}, [\![\epsilon]\!]_{\Sigma_\mathrm{o}})\}$,
- $\delta_D : M^D \times \Sigma_\mathrm{o} \to M^D$ *defined by* $\forall m \in M^D, \forall \sigma \in \Sigma_\mathrm{o} :$ $\delta_D(m, \sigma) \stackrel{\text{def}}{=} m \blacktriangleleft \mathrm{Obs}(\sigma)$.

A K-delay state estimator, for $\mathcal{G}$, is an LTS whose states contain suffixes of length K of "observable runs" that are compatible with the current observation on $\mathcal{G}$. On each transition fired by $\sigma \in \Sigma_\mathrm{o}$, possibly visited states more than K steps ago are forgotten, and the current state estimate is updated: for a transition, the arriving state is obtained using the shift operator ($\blacktriangleleft$) and putting in front (at the location of the current state estimate) compatible current states according to the state estimate at the previous step (i.e., $\mathrm{Obs}(\sigma)$ "filtered" by $\blacktriangleleft$).

D. K-step (state-based) Opacity

Opacity is defined on the observable and unobservable behaviors of the system. We consider that confidential information is directly encoded by means of a set of states $S \subseteq Q^\mathcal{G}$. If the current execution is $t \in \mathcal{L}(\mathcal{G})$, the attacker should not be able to deduce, from the knowledge of $P_{\Sigma_\mathrm{o}}(t)$ and the structure of $\mathcal{G}$, that the current state of the system is in S.

Confidentiality requirements may also prohibit inferring that the system went through a secret state in the past. To take into account this particularity, K-step opacity was introduced in [5][1]. Intuitively, K-step opacity takes into account the opacity of the secret in the past and also allows to say that the knowledge of the secret becomes worthless after the observation of a given number of actions.

Definition 3 (K-step opacity): For $K \in \mathbb{N}$, the secret S is K-step opaque on $\mathcal{G}$ under the projection P_{Σ_o} or $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K)$ opaque if

$$\forall t \in \mathcal{L}(\mathcal{G}), \forall t' \preceq t : |t - t'|_{\Sigma_\mathrm{o}} \leq K \wedge t' \in \mathcal{L}_S(\mathcal{G})$$
$$\Rightarrow \exists s \in \mathcal{L}(\mathcal{G}), \exists s' \preceq s : s \approx_{\Sigma_\mathrm{o}} t \wedge s' \approx_{\Sigma_\mathrm{o}} t' \wedge s' \notin \mathcal{L}_S(\mathcal{G}).$$

The secret S is K-step opaque on $\mathcal{G}$ if for every execution t of $\mathcal{G}$, for every secret execution t' prefix of t with an observable difference inferior to K, there exist two executions s and s' observationally equivalent respectively to t and t' s.t. s' is not a secret execution.

Remark 1: If S is $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K)$ opaque then S is $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K')$ opaque for $K' \leq K$. Moreover, 0-step opacity corresponds to "current" opacity (as defined in [2]).

Example 1: Consider $\Sigma_\mathrm{o} = \{a, b\}$.

[1]Compared with [5], for simplicity, we only consider a unique initial state and deterministic LTSs.

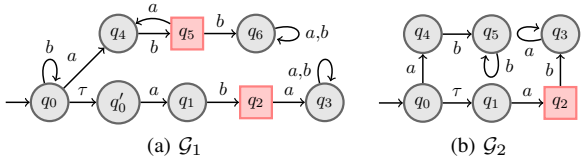


Fig. 2. Several systems with secret states (red squares)

- *On $\mathcal{G}_1$ (Fig. 2a), the secret is not $(\mathcal{G}_1, P_{\Sigma_\mathrm{o}}, 0)$-opaque, as after the observation of a trace in $b^* \cdot a \cdot b$, the attacker knows that the system is currently in a secret state (but does not know whether it is q_2 or q_5).*
- *On $\mathcal{G}_2$ (Fig. 2b), the secret is $(\mathcal{G}_2, P_{\Sigma_\mathrm{o}}, 1)$ opaque. However, the secret is not $(\mathcal{G}_2, P_{\Sigma_\mathrm{o}}, 2)$ opaque as only $\tau \cdot a \cdot b \cdot a$ is a compatible execution with the observation $a \cdot b \cdot a$. After the last a has occured, the attacker can deduce that the system was in state q_2 two steps ago.*

IV. Verification of Opacity at Runtime

A. Characterizing opacity on the observable behavior

To validate opacity with runtime techniques, we need to characterize K-step opacity in terms of observable behavior.

Proposition 1: $S \subseteq Q^\mathcal{G}$ is $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K)$ opaque iff

$$\forall \mu \in \mathcal{T}_{\Sigma_\mathrm{o}}(\mathcal{G}), \forall \mu' \preceq \mu : |\mu - \mu'| \leq K \Rightarrow [\![\mu'/\mu]\!]_{\Sigma_\mathrm{o}} \not\subseteq \mathcal{L}_S(\mathcal{G}).$$

S is $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K)$ opaque if for each observable system trace μ, and each of its prefixes μ' with less than K observations less than μ, if there exists an execution compatible with μ' ending in a secret state, then there exists another compatible execution that does not end in a secret state.

In the sequel, the set of traces, for which the K-step opacity of the secret is revealed, is formally defined by:

$$\mathrm{leak}(\mathcal{G}, P_{\Sigma_\mathrm{o}}, S) \stackrel{\text{def}}{=} \{\mu \in \mathcal{T}_{\Sigma_\mathrm{o}}(\mathcal{G}) \mid \exists \mu' \preceq \mu : |\mu - \mu'| \leq K \wedge [\![\mu'/\mu]\!]_{\Sigma_\mathrm{o}} \subseteq \mathcal{L}_S(\mathcal{G})\}. \quad (1)$$

Corollary 1: S is $(\mathcal{G}, P_{\Sigma_\mathrm{o}}, K)$ opaque iff $\mathrm{leak}(\mathcal{G}, P_{\Sigma_\mathrm{o}}, S) = \emptyset$.

In some cases, it is interesting to characterize the set of traces that reveal the secret at *exactly* k steps with $k \leq K$:

$$\mathrm{leak}(\mathcal{G}, P_{\Sigma_\mathrm{o}}, S, k) \stackrel{\text{def}}{=} \{\mu \in \mathcal{T}_{\Sigma_\mathrm{o}}(\mathcal{G}) \mid (3) \wedge (4)\}, \quad (2)$$

with

$$\exists \mu' \preceq \mu : |\mu - \mu'| = k \wedge [\![\mu'/\mu]\!]_{\Sigma_\mathrm{o}} \subseteq \mathcal{L}_S(\mathcal{G}), \quad (3)$$
$$\forall \mu' \preceq \mu : |\mu - \mu'| < k \Rightarrow [\![\mu'/\mu]\!]_{\Sigma_\mathrm{o}} \not\subseteq \mathcal{L}_S(\mathcal{G}). \quad (4)$$

That is, there exists an observation trace that reveals the opacity of the secret k steps ago (3) and every observation trace which is produced strictly less than k steps ago does not reveal the opacity (4). Furthermore, one may notice that $\bigcup_{0 \leq k \leq K} \mathrm{leak}(\mathcal{G}, P_{\Sigma_\mathrm{o}}, S, k) = \mathrm{leak}(\mathcal{G}, P_{\Sigma_\mathrm{o}}, S)$.

B. Synthesizing Runtime Verifiers

We present how we runtime verify the opacity of a secret on a given system using a monitor. A monitor captures, for each observation $\mu \in \Sigma^*_\mathrm{o}$, what the attacker can infer about the current execution of the system and the possible leakage of the secret w.r.t. the considered opacity. A monitor can be used by an administrator to discover opacity leakages on the system and take appropriate reactions.

Definition 4 (Runtime verifier): *A* runtime verifier *(R-Verifier) $\mathcal{V}$ is a finite-state machine $(Q^{\mathcal{V}}, q^{\mathcal{V}}_{\text{init}}, \Sigma_{\text{o}}, \delta_{\mathcal{V}}, \mathbb{D}, \Gamma^{\mathcal{V}})$ where $\Gamma^{\mathcal{V}} : Q^{\mathcal{V}} \to \mathbb{D}$ is the output function. $\mathbb{D} \stackrel{\text{def}}{=} \{\text{leak}_0, \ldots, \text{leak}_{\text{K}}, \text{noleak}\}$ is the truth domain.*

We now state the properties that an R-Verifier should satisfy:

Definition 5 (R-Verifier soundness and completeness): *An R-Verifier $\mathcal{V}$ is sound and complete w.r.t. $\mathcal{G}, P_{\Sigma_{\text{o}}}, S$ whenever $\forall \mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}), \forall l \in [0, K]$:*

$$\Gamma^{\mathcal{V}}(\delta_{\mathcal{V}}(q^{\mathcal{V}}_{\text{init}}, \mu)) = \text{leak}_{\text{l}} \quad \Leftrightarrow \quad \mu \in \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S, l)$$
$$\wedge\ \Gamma^{\mathcal{V}}(\delta_{\mathcal{V}}(q^{\mathcal{V}}_{\text{init}}, \mu)) = \text{noleak} \quad \Leftrightarrow \quad \mu \notin \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S).$$

An R-Verifier is sound ($\Rightarrow$ direction) if it never gives a false verdict. It is complete ($\Leftarrow$ direction) if all observations raise an appropriate "leak" verdict: a noleak verdict when the opacity is preserved, a leak_{l} verdict when the opacity leaks at l observable steps on the system. R-Verifiers are synthesized directly from K-delay state estimators.

Proposition 2: For $\mathcal{G}$, $S \subseteq Q^{\mathcal{G}}$, the R-Verifier $\mathcal{V} = (Q^{\mathcal{V}}, q^{\mathcal{V}}_{\text{init}}, \Sigma_{\text{o}}, \delta_{\mathcal{V}}, \mathbb{D}, \Gamma^{\mathcal{V}})$ built from the K-delay state estimator $D = (M^D, m^D_{\text{init}}, \Sigma_{\text{o}}, \delta_D)$ of $\mathcal{G}$ where $Q^{\mathcal{V}} = M^D, q^{\mathcal{V}}_{\text{init}} = m^D_{\text{init}}, \delta_{\mathcal{V}} = \delta_D$, and $\Gamma^{\mathcal{V}} : Q^{\mathcal{V}} \to \mathbb{D}$ defined by

- $\Gamma^{\mathcal{V}}(m) = \text{noleak}$ *if* $\forall k \in [0, K] : m(k) \notin 2^S$,
- $\Gamma^{\mathcal{V}}(m) = \text{leak}_{\text{l}}$ *where* $l = \min\{k \in [0, K] \mid m(k) \in 2^S\}$ *otherwise,*

is sound and complete w.r.t. $\mathcal{G}, P_{\Sigma_{\text{o}}}, S$ and K-step opacity.

V. Enforcement of opacity at runtime

We build runtime enforcers for K-step opacity. An underlying hypothesis is that the system is *live*, i.e., not deadlocked and always produces events, e.g., a reactive system. Roughly speaking, the purpose of a runtime enforcer is to read some (unsafe) execution sequence produced by $\mathcal{G}$ (input to the enforcer) and to transform it into an output sequence that is safe regarding opacity (see Fig. 1c).

A runtime enforcer acts as a delayer on an input sequence μ, using its internal memory to memorize some of the events produced by $\mathcal{G}$. It releases a prefix o of μ containing some stored events, when the system has produced enough events so that the opacity is ensured, i.e., when the enforcer releases an output o (the only sequence seen by the attacker), then either this sequence does not reveal the opacity of the secret or the system has already produced a sequence $\mu \succeq o$, making the knowledge of o obsolete to the attacker. For instance, if the enforcer releases a sequence o leaking the secret at $k \leq K$ steps, it has already received a sequence μ from the system s.t. $|\mu| - |o| > K - k$.

Let us illustrate informally how we enforce opacity.

Example 2 (Principle of enforcing opacity): On $\mathcal{G}_2$, the secret is not $(\mathcal{G}_2, P_{\Sigma_{\text{o}}}, 2)$ opaque because of the observation sequence $a \cdot b \cdot a$. A runtime enforcer will delay this sequence in such a way that, when the attacker determines that the system was in a secret secret, it is always more than $K = 2$ steps ago on the real system. That is, some of the events produced by the system will be retained inside the enforcer memory. Intuitively, for the aforementioned sequence, the expected behavior of a runtime enforcer is as follows.

Sequence of $\mathcal{G}_2$	*Obs. sequence*	*Memory*	*Output*
τ	ϵ	ϵ	ϵ
$\tau \cdot a$	a	ϵ	a
$\tau \cdot a \cdot b$	$a \cdot b$	ϵ	$a \cdot b$
$\tau \cdot a \cdot b \cdot a$	$a \cdot b \cdot a$	a	$a \cdot b$
$\tau \cdot a \cdot b \cdot a \cdot a$	$a \cdot b \cdot a \cdot a$	ϵ	$a \cdot b \cdot a \cdot a$
$\tau \cdot a \cdot b \cdot a^+$	$a \cdot b \cdot a^+$	ϵ	$a \cdot b \cdot a^+$

When the system produces the sequence $\tau \cdot a$, the enforcer should not modify the observation trace a which is safe regarding opacity. When the system produces the sequence $\tau \cdot a \cdot b$, the enforcer observes $a \cdot b$ and lets the system execute normally (we expect the system execution to be minimally modified). Then, when the system produces a new a, the enforcer memorizes this event (the attacker still sees $a \cdot b$). Next, when the system produces another a, the system was in a secret state 3 steps ago. Thus, the enforcer can release the first stored a. Indeed, when the attacker observes $a \cdot b \cdot a$, the system has produced $a \cdot b \cdot a \cdot a$, and was in the secret state q_2 three steps ago: $(\mathcal{G}_2, P_{\Sigma_{\text{o}}}, 2)$ opacity of S is thus preserved. Finally, the last received a and subsequent ones can be freely output by the enforcer since they preserve 2-step opacity.

A. Defining Runtime Enforcers

We define a generic notion of runtime enforcers which are special finite-state machines. By reading events, they produce enforcement operations that delay the input trace or release some already stored events to ensure opacity.

Definition 6 (Enforcement operations Ops and memory): *The memory of runtime enforcers is a list whose elements are pairs consisting of an observable event and an integer. The set of possible configurations of the memory is thus $\mathcal{M}(T) = \bigcup_{i=0}^{T} (\Sigma_{\text{o}} \times \mathbb{N})^i$. When an element $(\sigma, d) \in \Sigma_{\text{o}} \times \mathbb{N}$ is inside the memory, it means that the event σ has to be retained d units of time before being released by the enforcer to preserve opacity. Enforcement operations take as inputs an observable event and a memory content (i.e., a special sequence of events, detailed later) to produce in output an observable sequence and a new memory content: Ops $\subseteq 2^{(\Sigma_{\text{o}} \times \mathcal{M}(T)) \to (\Sigma^*_{\text{o}} \times \mathcal{M}(T))}$.*

Examples of enforcement operations consist of memorizing input events or halting the underlying system. In Section V-C, we define enforcement operations dedicated to opacity.

Definition 7 (Generic R-Enforcer (R-Enforcer(Ops))): *An R-Enforcer $\mathcal{E}$ is a 6-tuple $(Q^{\mathcal{E}}, q^{\mathcal{E}}_{\text{init}}, \Sigma_{\text{o}}, \delta_{\mathcal{E}}, Ops, \Gamma^{\mathcal{E}}, \mathcal{M}(T))$ defined relatively to a set of observable events Σ_{o} and parameterized by a set of enforcement operations Ops. The finite set $Q^{\mathcal{E}}$ denotes the control states, $q^{\mathcal{E}}_{\text{init}} \in Q^{\mathcal{E}}$ is the initial state. $\delta_{\mathcal{E}} : Q^{\mathcal{E}} \times \Sigma_{\text{o}} \to Q^{\mathcal{E}}$ is the transition function. The function $\Gamma^{\mathcal{E}} : Q^{\mathcal{E}} \to Ops$ associates an enforcement operation to each state.*

Informally an R-Enforcer performs a transition by reading an event produced by the system. The arriving state of the transition is associated to an enforcement operation which is applied to the current event and the memory content. In the following we abbreviate $\delta_{\mathcal{E}}(q, a) = q'$ by $q \xrightarrow{a}_{\mathcal{E}} q'$. The notion of *run* is naturally transposed from its definition for LTSs: for a trace $\mu = \sigma_0 \cdots \sigma_{n-1}$ of length n $\text{run}(\mu, \mathcal{E}) = (q_0, \sigma_0/\alpha_0, q_1) \cdot (q_1, \sigma_1/\alpha_1, q_2) \cdots (q_{n-1}, \sigma_{n-1}/\ \alpha_{n-1}, q_n)$,

with $\forall i \in [0, n-1] : \Gamma^{\mathcal{E}}(q_{i+1}) = \alpha_i$. In the remainder, $\mathcal{E} = (Q^{\mathcal{E}}, q^{\mathcal{E}}_{\text{init}}, \Sigma_{\text{o}}, \delta_{\mathcal{E}}, \mathit{Ops}, \Gamma^{\mathcal{E}}, \mathcal{M}(T))$ designates an R-Enforcer and $\mu \in \Sigma^*_{\text{o}}$ designates the current observation trace of the system input to the R-Enforcer. We formalize how R-Enforcers(Ops) react to input traces through the standard notions of *configuration* and *derivation*.

Definition 8 (Semantics of R-Enforcer(Ops)): A configuration *is a 3-tuple* $(q, \mu, c) \in Q^{\mathcal{E}} \times \Sigma^*_{\text{o}} \times \mathcal{M}(T)$ *where* q *denotes the current control state,* μ *the remaining trace to read, and* c *the current memory configuration.*

- *A configuration* (q', μ', c') *is* derivable in one step *from the configuration* (q, μ, c) *and* produces the output[2] $o \in \Sigma^*_{\text{o}}$, *and we note* $(q, \mu, c) \overset{o}{\hookrightarrow} (q', \mu', c')$ *if and only if* $\mu = \sigma \cdot \mu' \wedge q \xrightarrow{\sigma}_{\mathcal{E}} q' \wedge \Gamma^{\mathcal{E}}(q') = \alpha \wedge \alpha(\sigma, c) = (o, c')$.
- *A configuration* C' *is* derivable in several steps *from a configuration* C *and* produces the output $o \in \Sigma^*_{\text{o}}$, *and we note* $C \overset{o}{\Longrightarrow}_{\mathcal{E}} C'$, *if and only if*

$$\begin{aligned}\exists k \geq 0, \exists C_0, \ldots, C_k : \quad & C = C_0 \wedge C' = C_k \\ & \wedge \forall i \in [0, k[, \exists o_i \in \Sigma^*_{\text{o}} : C_i \overset{o_i}{\hookrightarrow} C_{i+1} \\ & \wedge o = o_0 \cdots o_{k-1}.\end{aligned}$$

We define the transformation performed by an R-Enforcer, with a set of enforcement operations Ops.

Definition 9 (Trace transformation): $\mathcal{E}$ *transforms* μ *into the output trace* $o \preceq \mu$ *as defined by the relation* $\Downarrow_{\mathcal{E}} \subseteq \Sigma^*_{\text{o}} \times \Sigma^*_{\text{o}}$, *where* ϵ *refers to* $\epsilon_{\Sigma_{\text{o}}}$*:*

- $\epsilon \Downarrow_{\mathcal{E}} \epsilon$,
- $\mu \Downarrow_{\mathcal{E}} o$ *if* $\exists q \in Q^{\mathcal{E}}, \exists c \in \mathcal{M}(T) : (q^{\mathcal{E}}_{\text{init}}, \mu, \epsilon_{\mathcal{M}}) \overset{o}{\Longrightarrow}_{\mathcal{E}} (q, \epsilon, c)$.

The empty sequence ϵ is not modified by $\mathcal{E}$ (i.e., when the system does not produce any event). The observation trace $\mu \in \Sigma^*_{\text{o}}$ is transformed by $\mathcal{E}$ into the trace $o \in \Sigma^*_{\text{o}}$, when the trace is transformed from the initial state of $\mathcal{E}$, starting with an empty memory. Note that the resulting memory configuration c depends on the sequence $\mu \setminus o$ of events read by the R-Enforcer but not produced in output yet as we shall see in the remainder of this section.

B. Enforcing the opacity at runtime

Before defining this enforcement notion more formally, we first formalize, for a given trace of $\mathcal{G}$, which of its prefixes can be safely output.

Definition 10 (Prefixes that are safe to output): For K-step opacity, a trace $\mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G})$, we say that it is safe to output $\mu' \preceq \mu$, noted $\text{safe}(\mu, \mu')$, if

$$\begin{aligned}&\mu' \notin \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S) \\ &\vee \exists k \leq K : (\mu' \in \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S, k) \wedge |\mu| - |\mu'| \geq K - k).\end{aligned}$$

That is, it is safe to produce $\mu' \preceq \mu$ if either μ' does not reveal the opacity or it reveals the opacity at k steps but it was produced on the system more than k steps ago. Note that when it is safe to produce a given trace, then all its prefixes are safe to produce:

$$\forall \mu, \mu' \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}) : \text{safe}(\mu, \mu') \Rightarrow \forall \mu'' \prec \mu' : \text{safe}(\mu, \mu'').$$

[2]Note that o can be ϵ if the enforcer chooses to not produce an output.

Furthermore, by convention, we will only consider systems for which it is safe to produce ϵ, i.e., when some sequences of $[\![\epsilon]\!]_{\Sigma_{\text{o}}}$ are not secret. Under the assumption that the system is alive, for a given trace, there always exists one of its extension traces which is safe to output, i.e.,

$$\forall \mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}), \exists \mu' \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}) : \mu \preceq \mu' \wedge \text{safe}(\mu', \mu).$$

e.g., any μ' s.t. $|\mu' - \mu| > K$. Moreover, the set of traces that lead a given sequence to be safe is extension-closed, i.e.,

$$\begin{aligned}\forall \mu' \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}) :& (\exists \mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}) : \mu' \preceq \mu \wedge \text{safe}(\mu, \mu')) \\ & \Rightarrow (\forall \mu'' \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}) : \mu \preceq \mu'' \Rightarrow \text{safe}(\mu'', \mu')).\end{aligned}$$

1) Expected properties for runtime enforcers.: We now explain what we mean exactly by *opacity enforcement*, and what are the consequences of this definition on the systems and secrets. The following constraints are expected to hold for the enforcers we aim to synthesize.

- *soundness*: the output trace should preserve the opacity of the system;
- *transparency*: the input trace should be modified *in a minimal way*, namely if it already preserves opacity it should remain unchanged, otherwise its *longest prefix* preserving opacity should be issued.

On Example 2, soundness entails a runtime enforcer to e.g., output $a \cdot b$ (instead of $a \cdot b \cdot a$) when $\mathcal{G}_2$ produces $\tau \cdot a \cdot b \cdot a$. Transparency entails a runtime enforcer to e.g., output $a \cdot b \cdot a \cdot a$ (instead of any prefix) when $\mathcal{G}_2$ produces $\tau \cdot a \cdot b \cdot a \cdot a$.

Remark 2: There always exists a trivial, sound but generally non transparent, enforcer delaying every event by K *units of time for* K*-step opacity.*

The formal definition of opacity-enforcement by a runtime enforcer relates the input sequence produced by the program fed to the enforcer and the allowed output sequence so that the enforcer is sound and transparent.

Definition 11 (Enforcement of opacity by an enforcer): The R-Enforcer $\mathcal{E}$ *enforces the* K*-step opacity of* S *w.r.t.* $P_{\Sigma_{\text{o}}}$ *on a system* $\mathcal{G}$ *if* $\forall \mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}), \exists o \preceq \mu : \mu \Downarrow_{\mathcal{E}} o \Rightarrow$

$$\mu \notin \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S) \Rightarrow o = \mu \quad (5)$$

$$\mu \in \text{leak}(\mathcal{G}, P_{\Sigma_{\text{o}}}, S) \Rightarrow o = \max_{\preceq}\{\mu' \preceq \mu \mid \text{safe}(\mu, \mu')\}. \quad (6)$$

A sound and transparent R-Enforcer always produces maximal safe sequences:

Proposition 3: For a sound and transparent R-Enforcer $\mathcal{E}$*:*

$$\begin{aligned}&\forall \mu \in \mathcal{T}_{\Sigma_{\text{o}}}(\mathcal{G}), \forall o \preceq \mu : \\ &\mu \Downarrow_{\mathcal{E}} o \Rightarrow \big(\text{safe}(\mu, o) \wedge \forall o \prec o' \preceq \mu : \neg \text{safe}(\mu, o')\big).\end{aligned}$$

Most of the previous enforcement approaches (e.g., [18], [14]) used enforcement mechanisms with a finite but unbounded memory under the soundness and transparency constraints. Since we are setting our approach in a general security context, we go one step further on the practical constraints expected for a desired enforcement mechanism dedicated to opacity. Here we consider that the memory allocated to the enforcer has a given size[3]:

- *do-not-overflow*: the size of the partial trace memorized by the enforcer does not exceed the allocated memory size.

[3]Besides memory size limitation, this constraint can represent the desired quality of service, e.g., maximal allowed delay.

2) When is the opacity of a secret enforceable on a system?: After stating the constraints on runtime enforcement for opacity, we need to delineate the systems, interfaces and secrets s.t. opacity is enforceable using runtime enforcers. Existence of sound and transparent R-Enforcers with unbounded memory for opacity relies first on the provided characterization of opacity preservation on the observable behavior as finitary properties (Section III-D) and second on existing results in enforcement monitoring of finitary properties (see e.g., [18], [14]). Now, the existence of an R-Enforcer for K-step opacity relies only on the *do-not-overflow* constraint of a memory of size T, defined as

$$\max_{\mu\in\mathcal{T}_{\Sigma_o}(\mathcal{G})}\left\{\min\{|\mu\setminus o| \mid o\preceq\mu\wedge \text{safe}(\mu,o)\}\right\}\leq T.$$

The previous enforcement criterion is not usable in practice and is not computable generally. Thus, we will give a more practical and decidable enforcement criterion using K-delay state estimators. To each state of the K-delay state estimator, we have seen that it is possible to determine the opacity leakage. Intuitively, the reasoning is as follows. If we reach a state in the K-delay state estimator s.t. it leaks the 2-step opacity of the secret (i.e., the attacker knows that the system was in a secret state 2 steps ago). Then for $K\geq 2$, the enforcer has to delay the last event produced by the system by $K-1$ units of time. Indeed, after that, the attacker will know that the system was in a secret state $K+1$ steps ago. This knowledge is safe w.r.t. K-step opacity.

The criterion on K-delay state estimators uses the following lemma, which is a direct consequence of the accuracy of state estimators.

Lemma 1: Given a system $\mathcal{G}$, a projection map P_{Σ_o}, and a secret S, the states of the K-delay state estimator $D=(Q^D,q^D_{\text{init}},\Sigma_o,\delta_D)$ are s.t.:

$$\begin{aligned}\forall\mu_1,\mu_2\in\mathcal{T}_{\Sigma_o}(\mathcal{G}):\delta_D(m^D_{\text{init}},\mu_1)=\delta_D(m^D_{\text{init}},\mu_2)\Rightarrow\\ \exists k\in[0,K]:\mu_1,\mu_2\in\text{leak}(\mathcal{G},P_{\Sigma_o},S,k)\\ \vee\ \mu_1,\mu_2\notin\text{leak}(\mathcal{G},P_{\Sigma_o},S).\end{aligned}$$

All traces ending in a given state of the state estimator reveal or preserve opacity in the same way. Thus, in a K-delay state estimator $D=(Q^D,m^D_{\text{init}},\Sigma_o,\delta_D)$, to each state $m\in Q^D$, we can associate the delay to hold (i.e., after which it is safe to "release") the last received event of the trace leading to this state in order to preserve opacity: $\forall m\in Q^D$:

$$\text{hold}(m)\ \stackrel{\text{def}}{=}\ \begin{cases}K+1-k & \text{when (7)}\\ 0 & \text{otherwise (i.e., when (8))}\end{cases}$$

with:

$$\begin{aligned}&\exists k\in[0,K],\forall\mu\in\mathcal{T}_{\Sigma_o}(\mathcal{G}):\\ &\qquad\delta_D(m^D_{\text{init}},\mu)=m\Rightarrow\mu\in\text{leak}(\mathcal{G},P_{\Sigma_o},S,k)\end{aligned}\tag{7}$$

$$\begin{aligned}&\forall\mu\in\mathcal{T}_{\Sigma_o}(\mathcal{G}):\\ &\qquad\delta_D(m^D_{\text{init}},\mu)=m\Rightarrow\mu\notin\text{leak}(\mathcal{G},P_{\Sigma_o},S)\end{aligned}\tag{8}$$

Equivalently, using an R-Verifier $\mathcal{V}=(Q^{\mathcal{V}},q^{\mathcal{V}}_{\text{init}},\Sigma_o,\delta_{\mathcal{V}},\mathbb{D},\Gamma^{\mathcal{V}})$ for $\mathcal{G}$ and K-step opacity, and synthesized from D (thus $Q^D=Q^{\mathcal{V}}$ and $q^D_{\text{init}}=q^{\mathcal{V}}_{\text{init}}$), $\forall\mu\in\mathcal{T}_{\Sigma_o}(\mathcal{G})$: $\text{hold}(\delta_D(m^D_{\text{init}},\mu))=K+1-k$ when $\Gamma^{\mathcal{V}}(\delta_{\mathcal{V}}(q^{\mathcal{V}}_{\text{init}},\mu))=\text{leak}_k$. Thus, synthesis of R-Enforcers will rely on the synthesis of R-Verifiers.

Proposition 4: Let $D=(Q^D,m^D_{\text{init}},\Sigma_o,\delta_D)$ be the K-delay state estimator associated to $\mathcal{G}$. The K-step opacity of the secret S is enforceable by an R-Enforcer with a memory of size T iff $\max\{hold(m)\mid m\in Q^D\}\leq T$.
Consequently, enforcement of a K-step opacity with a memory of a given size is decidable.

C. Synthesis of runtime enforcers

To address the synthesis of runtime enforcers for opacity, we first define their primitives: the enforcement operations.

1) Enforcement operations: Let us define some auxiliary operations. In the following, we will use the following notations for the memory of R-Enforcers. For a pair $(\sigma,d)\in\Sigma_o\times\mathbb{N}$, $(\sigma,d).delay\stackrel{\text{def}}{=}d$. For two memory configurations c,c' s.t. $c=((\sigma_1,d_1)\cdots(\sigma_t,d_t))$, $c'=((\sigma_1,d_1)\ \cdots(\sigma_{t'},d_{t'}))$ with $t'\leq t$,:

- $c_{\downarrow\Sigma_o}\stackrel{\text{def}}{=}\sigma_1\cdots\sigma_t$,
- $(c\setminus c')_{\downarrow\Sigma_o}$ is ϵ_{Σ_o} if $c=c'$ and the sequence of events $\sigma_{t'+1}\cdots\sigma_t$ otherwise.

Definition 12 (Auxiliary operations): For a memory $\mathcal{M}$ of size T, given $t\leq T$ and $c=(\sigma_1,d_1)\cdots(\sigma_t,d_t)\in\mathcal{M}(T)$, free *and* delay *($\mathcal{M}(T)\to\mathcal{M}(T)$) are defined as follows:*

- $\text{delay}(c)\stackrel{\text{def}}{=}(\sigma_1,d_1-1)\cdots(\sigma_t,d_t-1)$*, with $t\leq T$;*
- $\text{free}(c)\stackrel{\text{def}}{=}(\sigma_i,d_i)\cdots(\sigma_t,d_t)$*, with $1\leq i\leq t$ and*

$\forall j\in[1,i-1]:c^j.delay\leq 0\ \wedge\ \forall j\in[i,t]:c^j.delay>0$.
The operation delay consists in decrementing the delay of each element inside the memory. Intuitively, this operation is used when one step has been performed on the system, and thus the stored events revealing the opacity have to be retained for one unit of time less. The operation free consists in outputting the events that currently do not leak opacity (with a negative or null delay). The following operations are those actually used by the runtime enforcers.

Definition 13: The enforcement operations are defined as follows where $\sigma\in\Sigma_o$, $c=(\sigma_1,d_1)\cdots(\sigma_t,d_t)\in\mathcal{M}(T)$.

- $\text{store}_d(\sigma,c)\stackrel{\text{def}}{=}(o,c'\cdot(\sigma,d))$*, with* $c'=\text{free}\circ\text{delay}(c)$, $o=(c\setminus c')_{\downarrow\Sigma_o}$*;*
- $\text{dump}(\sigma,c)\stackrel{\text{def}}{=}(o,c'')$ *with*
  - $c'\stackrel{\text{def}}{=}\text{free}\circ\text{delay}(c)$,
  - $o\stackrel{\text{def}}{=}c_{\downarrow\Sigma_o}\cdot\sigma$ *if* $c'=\epsilon_{\mathcal{M}}$ *and* $(c\setminus c')_{\downarrow\Sigma_o}$ *else,*
  - $c''\stackrel{\text{def}}{=}(c\setminus c')\cdot(\sigma,0)$ *if* $c'\neq\epsilon_{\mathcal{M}}$ *and* $\epsilon_{\mathcal{M}}$ *else;*
- $\text{off}(\sigma,c)\stackrel{\text{def}}{=}\text{dump}(\sigma,c)$*;*
- $\text{halt}(\sigma,c)\stackrel{\text{def}}{=}(\epsilon_{\Sigma_o},\epsilon_{\mathcal{M}})$.

For $d\in[1,K]$, the store_d operation is issued when the event submitted to the R-Enforcer should be delayed by d unit(s) of time in order to preserve opacity. This operation consists in first releasing the events preserving the opacity (using free$\circ$delay) and appending the event with the needed delay to the memory. The dump operation is issued when the submitted event does not reveal the opacity. The event is submitted but not necessarily produced in output. The R-Enforcer first releases the events preserving the opacity. After this step, if the memory is empty, then the event is appended to the output sequence. Otherwise, the event is appended in the memory with delay 0 so as to first be released in

the future and preserve the order of the input trace. The off operation is issued by an R-Enforcer when the opacity will not be revealed whatever are the future observable events produced by the system. Thus, the R-Enforcer can be switched off. Although the off has the same definition as the dump operation, such an enforcement operation is useful in practice since it reduces the overhead induced by the R-Enforcer. The halt operation is issued when the considered notion of opacity is irremediably revealed. This operation consists in ignoring the submitted event, erasing the memory, and stopping the underlying system.

2) Synthesis of R-Enforcers: We propose now to address the synthesis of R-Enforcers relying on K-delay state estimators and the function hold.

Proposition 5: Given $\mathcal{G}$, S and $\Sigma_o \subseteq \Sigma$, the R-Enforcer $\mathcal{E} = (Q^{\mathcal{E}}, q^{\mathcal{E}}_{\text{init}}, \Sigma_o, \delta_{\mathcal{E}}, \{\text{halt}, \text{store}_d, \text{dump}, \text{off} \mid d \in [1,K]\}, \Gamma^{\mathcal{E}}, \mathcal{M}(T))$, *built from the K-delay state estimator* $D = (M^D, m^D_{\text{init}}, \Sigma_o, \delta_D)$ *of $\mathcal{G}$ where* $Q^{\mathcal{E}} = M^D, q^{\mathcal{E}}_{\text{init}} = m^D_{\text{init}}, \delta_{\mathcal{E}} = \delta_D$, *and* $\Gamma^{\mathcal{E}} : Q^{\mathcal{E}} \to \{\text{off}, \text{dump}, \text{store}_d, \text{halt} \mid d \in [1,T]\}$ *defined by:*

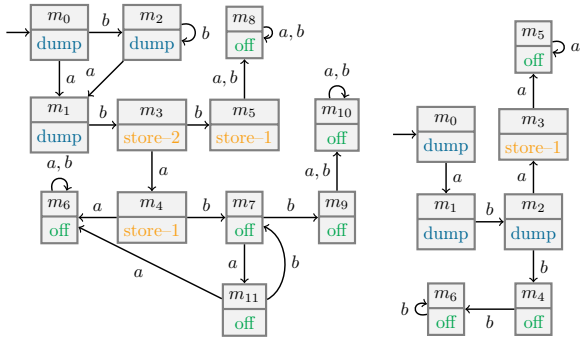
- $\Gamma^{\mathcal{E}}(m) = \text{off}$ *if* $\text{hold}(m) = 0 \wedge \forall m' \in \Delta_D(\{m\}, \Sigma_o^*) : \text{hold}(m') = 0$,
- $\Gamma^{\mathcal{E}}(m) = \text{dump}$ *if* $\text{hold}(m) = 0 \wedge \exists m' \in \Delta_D(\{m\}, \Sigma_o^*) : \text{hold}(m') \neq 0$,
- $\Gamma^{\mathcal{E}}(m) = \text{store}_d$ *if* $\exists d \in [1,T] : \text{hold}(m) = d$,
- $\Gamma^{\mathcal{E}}(m) = \text{halt}$ *if* $\exists d > T : \text{hold}(m) = d$.

enforces the K-step opacity of S under P_{Σ_o} on $\mathcal{G}$.

An R-Enforcer built following this construction processes an observation trace of the underlying system and enforces the opacity of the secret. As for the R-Verifier, the R-Enforcer has the same structure as the K-delay state estimator. To build the R-Enforcer, for each state m of the K-delay state estimator, one needs to determine $\text{hold}(m)$ which can be obtained using the opacity leakage provided by the R-Verifier (see Section V-B). An R-Enforcer switches off when the current read observation trace and all its possible continuations on the system do not leak the opacity of the secret. It dumps the event of the last observed trace when this trace does not leak the opacity but there is a possibility that the secret may leak in the future. It stores the last event of the observed trace in memory for d unit(s) of time, with $d \leq T$, when the current sequence leads the K-step opacity to be revealed $K+1-d$ step(s) ago ($\text{hold}(m) = d$). Consequently, when the attacker will observe this event, this will reveal the opacity of the secret at strictly more than K steps. When the current sequence leaks the K opacity of the secret d steps ago with $d > T$, the R-Enforcer halts the underlying system since the last event of this sequence cannot be memorized with the allocated memory of size T.

Example 3: Fig. 3 represents the R-Enforcers of $\mathcal{G}_1$ and $\mathcal{G}_2$ for 1-step and 2-step opacity, respectively. We assume a sufficient memory, i.e., $T \geq 2$ for $\mathcal{G}_1$ and $T \geq 1$ for $\mathcal{G}_2$.

Remark 3 (R-Enforcer optimization): In R-Enforcers, we can reduce the states in which the off *operation is produced, into a unique state. This is a straightforward adaptation of the transformation that is not modifying their correctness.*



(a) 1-step of $\mathcal{G}_1$ (b) 2-step of $\mathcal{G}_2$

Fig. 3. R-Enforcer for K-step opacity

VI. Related work

A. Model-checking of K-step opacity

In [19] and its companion paper [20], the authors addressed the model-checking of K-step opacity, using K-delay state estimators. A secret S is $(\mathcal{G}, P_{\Sigma_o}, K)$ opaque iff there does not exist a state m reachable in D, the K-delay state estimator of $\mathcal{G}$, such that $\exists k \in [0,K] : m(k) \in 2^S$.

B. Using runtime techniques

1) Validating simple opacity via testing: In [21], the authors are interested in testing the simple opacity of a system. In the context of ensuring opacity via access control mechanisms, the authors extend the classical theory of conformance testing in order to derive tests that detect violation of conformance by an access control implementation to its specification. Testing is another runtime-based validation technique to validate opacity. The authors only address the current opacity of the secret. Validation of the K-step based opacity through testing remains to be studied.

2) Runtime verification and enforcement for linear-time properties: Numerous runtime verification and enforcement frameworks exist for linear-time properties (cf. [17] for a short survey). Most of them focus on monitoring safety properties. Runtime enforcement was initiated by the work of [15] on what has been called *security automata*; i.e., monitors watching the execution and halting the program whenever it deviates from the desired property. Later, [22] proposed a more powerful enforcement mechanism called *edit-automata*. This mechanism featured the idea of "suppressing" (i.e., freezing) and "inserting" (frozen) actions in the current execution of a system.

To the best of our knowledge, only one runtime validation approach was proposed for (current) opacity [23]. Thus, this article first addresses runtime verification for K-step opacity and introduces runtime enforcement as a validation technique for opacity. Note that the notion of runtime enforcer proposed in this paper is inspired from and extends the variant used in [24] to enforce linear-time properties.

C. Comparison with supervisory control

Because of the halt operation, runtime enforcement is similar to supervisory control. Indeed, blocking the system or letting its execution going through are the only primitives endowed to controllers. The difference between supervisory

control and runtime enforcement is as follows. In supervisory control, the underlying system is put in parallel with a controller. When a controlled system tries to perform an illegal action, this action is disabled by the controller. In runtime enforcement, actions of the systems are directly fed to the enforcer, that delays or suppresses illegal actions. Illegal actions are thus actually executed on the system but not produced in output. (the effect of illegal actions is not visible from the outside). Hence, enforcement monitoring is appropriate to ensure a desired behavior on the system outputs, while supervisory control is appropriate to ensure a desired behavior on the internal behavior of a system. Finally, note that, the system to be considered in a runtime enforcement approach is the initial system along with its runtime enforcer, while in supervisory control, it is the product between the initial system and the controller.

VII. Conclusion and future work

1) Conclusion: We are interested in the use of runtime techniques to ensure K-step opacity. Proposed runtime techniques are complementary to supervisory control, which is usually used to validate opacity on systems. With runtime verification, we are able to detect leakages for the various levels of opacity. With runtime enforcement, opacity leakages are prevented, and this technique guarantees opacity preservation for the system of interest. The techniques proposed in this paper have several advantages compared to existing validation approaches for opacity. With the aim of ensuring the opacity of system, runtime enforcement is a non intrusive technique that is not damaging the internal nor the observable behavior of the underlying system. All results are implemented in a toolbox, named TAKOS: `http://toolboxopacity.gforge.inria.fr`.

2) Future work: Other opacity conditions could be handled in this framework such as *initial opacity* (cf. [25]) or *infinite-step opacity* (cf. [26]). New kinds of state estimators are needed, as shown in [25], [26] for verification purposes. As the proposed runtime techniques are complementary to supervisory control, we plan to study how we can combine those techniques to obtain the best of both worlds. For instance, when runtime enforcement with a given memory size is not possible, one may be interested in synthesizing controllers to restrict the system so as to ensure the existence of enforcers. Two practical implementation issues should be addressed. The first one is the retrieval of a suitable model of the analyzed system from its source or binaries for the proposed techniques to be applicable. The second one concerns integrating/translating the synthesized (high-level) verifiers and enforcers to general-purpose programming languages.

References

[1] Y. Falcone and H. Marchand, "Various notions of opacity verified and enforced at runtime," INRIA, Tech. Rep. 7349, 2010.

[2] E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau, "Concurrent secrets," *Discrete Event Dynamic Systems*, vol. 17, no. 4, pp. 425–446, 2007.

[3] J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan, "Opacity generalised to transition systems," *Int. J. of Information Security*, vol. 7, no. 6, pp. 421–435, 2008.

[4] R. Alur and S. Zdancewic, "Preserving secrecy under refinement," in *33rd Internat. Colloq. on Automata, Languages and Programming*, ser. LNCS, vol. 4052, 2006, pp. 107–118.

[5] A. Saboori and C. N. Hadjicostis, "Verification of k-step opacity and analysis of its complexity," *IEEE Trans. Automation Science and Engineering*, vol. 8, no. 3, pp. 549–559, 2011.

[6] J. Dubreil, P. Darondeau, and H. Marchand, "Supervisory control for opacity," *IEEE Trans. on Automat. Contr.*, vol. 55, no. 5, pp. 1089–1100, 2010.

[7] S. Takai and Y. Oka, "A formula for the supremal controllable and opaque sublanguage arising in supervisory control," *SICE J. of Contr., Measurement, and System Integration*, vol. 1, no. 4, pp. 307–312, March 2008.

[8] S. Takai and R. Kumar, "Verification and synthesis for secrecy in discrete-event systems," in *American Contr. Conf.*, 2009, pp. 4741–4746.

[9] A. Saboori and C. N. Hadjicostis, "Opacity-enforcing supervisory strategies via state estimator constructions," *IEEE Trans. Automat. Contr.*, vol. 57, no. 5, pp. 1155–1165, 2012.

[10] F. Cassez, J. Dubreil, and H. Marchand, "Dynamic observers for the synthesis of opaque systems," in *7th Int. Symposium on Automated Technology for Verification and Analysis*, 2009, pp. 352–367.

[11] Y. Wu and S. Lafortune, "Enforcement of opacity properties using insertion functions," in *51st IEEE Conf. on Decision and Contr.*, 2012, pp. 6722–6728.

[12] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Int. Symp. on Formal Methods*, 2006, pp. 573–586.

[13] K. Havelund and A. Goldberg, "Verify your runs," in *Verified Software: Theories, Tools, Experiments: 1st IFIP TC 2/WG 2.3 Conf., Revised Selected Papers and Discussions*, 2008, pp. 374–383.

[14] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties," in *9th Work. on Runtime Verification*, 2009, pp. 40–59.

[15] F. B. Schneider, "Enforceable security policies," *ACM Trans. of Information System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[16] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Programming Lang. and Syst.*, vol. 28, no. 1, pp. 175–205, 2006.

[17] Y. Falcone, "You should better enforce than verify," in *1st Int. Conf. on Runtime Verification*, ser. LNCS, vol. 6418, 2010, pp. 89–105.

[18] J. Ligatti, L. Bauer, and D. Walker, "Enforcing Non-safety Security Policies with Program Monitors," in *European Symposium on Research in Computer Security*, 2005, pp. 355–373.

[19] A. Saboori and C. N. Hadjicostis, "Verification of k-step opacity and analysis of its complexity," in *48th IEEE Conf. Decision and Contr.*, 2009, pp. 5056–5061.

[20] ——, "Delayed state estimation in discrete event systems and applications to security problems," UIUC Coordinated Science Laboratory, Tech. Rep., February 2008.

[21] H. Marchand, J. Dubreil, and T. Jéron, "Automatic testing of access control for security properties," in *21th IFIP Int. Conf. on Testing of Communicating Systems*, ser. LNCS, vol. 5826, 2009, pp. 113–128.

[22] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of non-safety policies," *ACM Trans. on Information and System Security*, vol. 12, no. 3, pp. 1–41, 2009.

[23] J. Dubreil, T. Jéron, and H. Marchand, "Monitoring confidentiality by diagnosis techniques," in *European Contr. Conf.*, 2009, pp. 2584–2590.

[24] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *Formal Meth. in Syst. Design*, vol. 38, no. 3, pp. 223–262, 2011.

[25] A. Saboori and C. N. Hadjicostis, "Verification of initial-state opacity in security applications of discrete event systems," *Inf. Sci.*, vol. 246, pp. 115–132, 2013.

[26] ——, "Verification of infinite-step opacity and complexity considerations," *IEEE Trans. Automat. Contr.*, vol. 57, no. 5, pp. 1265–1269, 2012.